Types for controlling heap and stack in Java

Emmanuel Hainry, Romain Péchoux

Types for controlling heap and space in Java

Emmanuel Hainry and Romain Péchoux

Université de Lorraine, LORIA, UMR 7503, Nancy, France
{hainry,pechoux}@loria.fr

Abstract. A type system is introduced for a strict but expressive subset of Java in order to infer resource upper bounds on both the heap-space and the stack-space requirements of typed programs. This type system is inspired by previous works on Implicit Computational Complexity, using tiering and non-interference techniques. The presented methodology has several advantages. First, it provides explicit polynomial upper bounds to the programmer, hence avoiding OutOfMemory and StackOverFlow errors. Second, type checking is decidable in linear time. Last, it has a good expressivity as it analyzes most object oriented features like overload, inheritance, and also handles flow statements controlled by objects.

1 Introduction

In the last decade, the development of embedded systems and mobile computing has led to a renewal of interest in predicting program resource consumption. This kind of problematic is highly challenging for popular object oriented programming languages which come equipped with environments for applications running on mobile and other embedded devices (e.g. Dalvik, Java ME or Java Card).

The current paper tackles such an issue by introducing a type system for a compile-time analysis of both heap and stack space requirements of Java-like programs thus avoiding OutOfMemory and StackOverFlow errors, respectively. The analyzed programs form a strict but expressive subset of Java, named core Java which features traits like recurrence, while loops, inheritance, override, overload. Core Java will be presented in a theoretically oriented manner in order to highlight the theoretical soundness of our results. It can be seen as a language strictly more expressive than Featherweight Java [18] enriched with features like variable updates and while loops.

The type system combines ideas coming from tiering discipline, used for complexity analysis of function algebra [3, 21], together with ideas coming from non-interference, used for secure information flow analysis [26]. It is inspired by two previous works: the seminal paper [22], initiating imperative programs type-based complexity analysis using secure information flow, which provides a characterization of polynomial time computable functions; and the paper [12], extending previous analysis to C processes with a fork/wait mechanism, which provides a characterization of polynomial space computable functions, but this work differs on a number of points. First, it is an extension to the object-oriented paradigm (although imperative feature can be dealt with). In particular, it

characterizes the complexity of recursive and non-recursive method calls whereas previous works where restricted to while loops. Second, it studies program intensional properties (like heap and stack) whereas previous papers were focusing on the extensional part (characterizing function spaces). Consequently, it is closer to a programmer's expectations. Third, it provides explicit big O polynomial upper bounds while the two aforementioned studies were only certifying algorithms to compute a function belonging to some fixed complexity class.

In our setting, the heap is represented by a directed graph where nodes are object addresses and arrows map an object address to its attribute addresses. The type system splits variables in two tiers: tier **0** and tier **1**. While tier **1** variables are pointers to nodes of the initial heap, tier **0** variables may point to newly created addresses. Information may flow from tier **1** to tier **0**.But our type system precludes flows from **0** to **1**. Naively, tier **1** variables are the ones that can be used either as guards of a while loop or as a recursive argument in a method call whereas tier **0** variables are just used as a storage for computed data. The idea for the polynomial upper bound relies on the fact that if the input graph structure has size n then the number of distinct possible configurations for k tier **1** variables is at most $O(n^k)$.

There are several related works on the complexity of imperative and object oriented languages. On imperative languages, the papers [24, 23, 19] study theoretically the heap-space complexity of core-languages using type systems based on a matrix calculus. On OO programming languages, the papers [14, 15] control the heap-space consumption using type systems based on amortized complexity introduced in previous works on functional languages [13, 20, 6]. Though similar, our result differs on several points.First, our analysis is not restricted to linear heap-space upper bounds. Second, it also applies to stack-space upper bounds. Last but not least, our language is not restricted to the expressive power of method calls and includes a `while` statement, controlling the interlacing of such a purely imperative feature with functional features like recurrence being a very hard task from a complexity perspective. Another interesting line of research is based on the analysis of heap-space and time consumption of Java bytecode [1, 2, 7]. The results from [1, 2] make use of abstract interpretations to infer efficiently symbolic upper bounds on resource consumption of Java programs. A constraint-based static analysis is used in [7] and focuses on certifying memory bounds for Java Card. Our analysis can be seen as a complementary approach since we try to obtain practical upper bounds through a cleaner theoretically oriented treatment. Consequently, this approach allows us to deal with our typing discipline on the original Java code without considering the corresponding Java bytecode. A complex type-system that allows the programmer to verify linear properties on heap-space is presented in [8]. Our result in contrast presents a very simple type system that however guarantees a polynomial bound.

In a similar vein, characterizing complexity classes below polynomial time is studied in [16, 17]. This work relies on a programming language called PURPLE combining imperative statements together with pointers on a fixed graph structure. Although not directly related, our type system was inspired by this work.

This work is independent from termination analysis but our main result relies on such analysis. Indeed, the polynomial bounds on both the stack and the heap of typed programs provided by Theorem 1 only hold for a terminating computation. Consequently, our analysis can be combined with termination analysis in order to certify the upper bounds on any input. Possible candidates for the imperative fragment are *Size Change Termination* [4, 5], tools like Terminator [9] based on *Transition predicate abstraction* [25] or symbolic complexity bound generation based on abstract interpretations, see [10, 11] for example.

2 Core Java syntax

We start to introduce the syntax of the considered core Java language a strict but expressive subset of Java. Expressions, instructions, methods, constructors and classes are defined by the grammar of Figure 1, with $\mathtt{x} \in \mathbb{V}$, $op \in \mathbb{O}$, $\mathrm{C} \in \mathbb{C}$, $m \in$

$E ::= \mathtt{x} \mid \mathtt{null} \mid \mathtt{this} \mid \mathtt{true} \mid \mathtt{false} \mid op(\overline{E}) \mid \mathtt{new}\ \mathrm{C}(\overline{E}) \mid E.m(\overline{E})$

$I ::= ; \mid [\tau]\ \mathtt{x}{:=}E; \mid I_1\ I_2 \mid \mathtt{while}(E)\{I\} \mid \mathtt{if}(E)\{I_1\}\mathtt{else}\{I_2\} \mid E.m(\overline{E});$

$M_{\mathrm{C}} ::= \tau\ m(\tau_1\ \mathtt{x}_1, \ldots, \tau_n\ \mathtt{x}_n)\{I[\mathtt{return}\ \mathtt{x};]\}$

$K_{\mathrm{C}} ::= \mathrm{C}(\tau_1\ \mathtt{y}_1, \ldots, \tau_n\ \mathtt{y}_n)\{\mathtt{x}_1{:=}\mathtt{y}_1; \ldots \mathtt{x}_n{:=}\mathtt{y}_n;\}$

$\mathfrak{C} ::= \mathrm{C}\{\tau_1\ \mathtt{x}_1; \ldots; \tau_n\ \mathtt{x}_n;\ K_{\mathrm{C}}\ M_{\mathrm{C}}^1 \ldots M_{\mathrm{C}}^k\}$

Fig. 1: Syntax of core Java

$\mathbb{M}$, $\mathbb{V}$ being the set of variables, $\mathbb{O}$ of operators, $\mathbb{M}$ of method names and $\mathbb{C}$ of class names. The τs are type annotations in $\mathbb{C} \cup \{\mathtt{void}, \mathtt{boolean}\}$. $[e]$ denotes some optional element e and $\overline{E}$ denotes a sequence of expressions $E_1, \ldots, E_n$. ';' denotes the empty instruction. This syntax does not include a `for` instruction as it can be simulated with a `while`. Also notice that there is no attribute access in our syntax using the '.' operator. Getters will be needed, as if all attributes were `private`. On the opposite, methods and classes are `public`.

Definition 1. *A core Java* program *is a collection of classes together with exactly one executable: Exe*$\{\mathtt{main}()\{\tau_1\ \mathtt{x}_1 := E_1; \ldots; \tau_n\ \mathtt{x}_n := E_n; I\}\}$.

In an executable, the instruction $\tau_1\ \mathtt{x}_1 := E_1; \ldots; \tau_n\ \mathtt{x}_n := E_n$; *is called the* initialization instruction *whereas* I *is called the* computational instruction.

We adopt OO nomenclature regarding attributes, parameters, signatures, and local variables. Also let $\mathrm{C}.\mathcal{A}$ denote the set of the attributes of the class. We write $m \in \mathrm{C}$ to denote that the method name m is declared in C.

Throughout the paper, we assume that programs are well-formed, meaning that there does not appear undefined class names or variables, no multiply

declared variables, no name clashes (variable names and class names are unique), signatures of methods are respected by the implementations.

3 Core Java pointer graph semantics

In this section, we provide a pointer graph semantics of core Java programs for representing the memory heap.

3.1 Pointer graph

Definition 2. *A* pointer graph $\mathcal{G}_\mathcal{P}$ *is a directed graph* $\mathcal{G} = (V, A)$ *together with a mapping* $\mathcal{P}$*. The nodes in* V *are references labeled by class names and the arrows in* A *link to references to the attributes and are labeled by the attribute name. Let* l *be the node label mapping from* V *to* $\mathbb{C}$ *and* i *be the arrow label mapping from* A *to* $\cup_{C\in\mathbb{C}} C.\mathcal{A}$*. The partial mapping* $\mathcal{P} : \mathbb{V} \cup \{\texttt{this}\} \mapsto V$ *is called a* pointer mapping*. Let* $dom(\mathcal{P})$ *to be domain of* $\mathcal{P}$*.*

This graph explicits the arborescent nature of objects: each constructor call creates a new node with arrows to its attributes, thus implementing the dynamic binding principle. Fig. 2 illustrates the pointer graph associated to a sequence of instantiations. Both the graph and the pointer mapping symbolized by snake arrows are represented.

```
B b := new B(new A(), new A());
C c := new C(b);
D d := new D(c);
B e := new B(c, c);
```

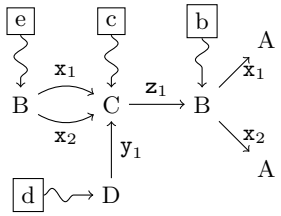

Fig. 2: Example of a pointer graph

3.2 Pointer stack

When calling a method, references to the parameters are pushed on the pointer stack which contains pointer mappings.

Definition 3. *A* pointer stack $\mathcal{S}_\mathcal{G}$ *is a LIFO structure of pointer mappings* $\mathcal{S}$ *corresponding to the same directed graph* $\mathcal{G}$*. Given a pointer stack* $\mathcal{S}_\mathcal{G}$*, define* $\top\mathcal{S}$ *to be the top pointer mapping of* $\mathcal{S}$*.*

Intuitively, the pointer mappings of a pointer stack $\mathcal{S}_\mathcal{G}$ map method parameters to the references of the arguments on which they are applied.

3.3 Memory configuration

A *primitive store* σ is a partial mapping $\sigma : \mathbb{V} \mapsto \{\texttt{true}, \texttt{false}\}$ associating a boolean value to some variable of primitive data type in $\mathbb{V}$.

Definition 4. *A* memory configuration $\mathcal{C}$ *is a quadruple* $\langle \mathcal{G}, \mathcal{P}, \mathcal{S}, \sigma \rangle$ *such that* $\mathcal{G}_\mathcal{P}$ *is a pointer graph,* $\mathcal{S}_\mathcal{G}$ *is a pointer stack and* σ *is a primitive store.*
The initial configuration $\mathcal{C}_0$ *is defined by* $\mathcal{C}_0 = \langle (\{\&\texttt{null}\}, \emptyset), \emptyset, [], \emptyset \rangle$ *where* $\emptyset$ *is used both for empty set and empty mapping,* $[]$ *denotes the empty pointer stack, and* $\&\texttt{null}$ *is the reference of the* `null` *object.*

3.4 Meta-language and flattening

The semantics of core Java programs will be defined on a meta-language with flat expressions and instructions, and special meta-instructions `push` and `pop`.

$$
\begin{array}{rl}
ME ::= & \texttt{x} \mid \texttt{null} \mid \texttt{this} \mid \texttt{true} \mid \texttt{false} \mid op(\texttt{x}_1, \ldots, \texttt{x}_n) \mid \\
& \texttt{new C}(\texttt{x}_1, \ldots, \texttt{x}_n) \mid \texttt{y}.m(\texttt{x}_1, \ldots, \texttt{x}_n) \\
MI ::= & \; ; \mid [\tau]\ \texttt{x}{:=}ME; \mid MI_1\ MI_2 \mid \texttt{x}.m(\texttt{y}_1, \ldots, \texttt{y}_n); \mid \\
& \texttt{while(x)}\{MI\} \mid \texttt{if(x)}\{MI_1\}\texttt{else}\{MI_2\} \mid \texttt{pop}; \mid \texttt{push}(\mathcal{P}); \mid \epsilon
\end{array}
$$

Flattening an instruction I into a meta-instruction $\overline{I}$ will consist in adding fresh intermediate variables for each complex parameter. This procedure is standard and does not change the semantics.

Lemma 1. *Define the size* $|I|$ *of an instruction* I *(respectively meta-instruction* $|MI|$*) to be the number of symbols in* I *(resp.* MI*). For each instruction* I*, we have* $|\overline{I}| = O(|I|)$.

3.5 Program semantics

Informally, the small step semantics $\rightarrow$ of core Java relates a pair $(\mathcal{C}, MI)$ of memory configuration $\mathcal{C}$ and meta-instruction MI to another pair $(\mathcal{C}', MI')$. Let $\rightarrow^*$ (respectively $\rightarrow^+$) be its reflexive and transitive (respectively transitive) closure. In the special case where $(\mathcal{C}, MI) \rightarrow^* (\mathcal{C}', \epsilon)$, we say that the meta-instruction MI *terminates on memory configuration* $\mathcal{C}$.

Definition 5. *A program of executable* $Exe\{\texttt{main}()\{\tau_1\ \texttt{x}_1 := E_1; \ldots; \tau_n\ \texttt{x}_n := E_n; I\}\}$ *terminates if the following conditions hold:*

1. $(\mathcal{C}_0, \overline{\tau_1\ \texttt{x}_1 := E_1; \ldots; \tau_n\ \texttt{x}_n := E_n;}) \rightarrow^* (\mathcal{C}, \epsilon)$
2. $(\mathcal{C}, \overline{I}) \rightarrow^* (\mathcal{C}', \epsilon)$

The memory configuration $\mathcal{C}$ *is called the* input.

Given a memory configuration $\mathcal{C} = \langle \mathcal{G}, \mathcal{P}, \mathcal{S}, \sigma \rangle$, let $\mathcal{C}(\texttt{x})$, intuitively the value of `x`, be defined by: $\mathcal{C}(\texttt{x}) = \sigma(\texttt{x})$ if $\texttt{x} \in dom(\sigma); \mathcal{C}(\texttt{x}) = \top\mathcal{S}(\texttt{x})$ if $\texttt{x} \in dom(\top\mathcal{S}); \mathcal{C}(\texttt{x}) = \mathcal{P}(\texttt{x})$ if $\texttt{x} \in dom(\mathcal{P})$ and let $\mathcal{C}[\mu : \texttt{x} \mapsto v]$, $\mu \in \{\sigma, \mathcal{P}, \top\mathcal{S}\}$, be a notation for the memory configuration $\mathcal{C}'$ that is equal to $\mathcal{C}$ but on μ where $\mathcal{C}'(\texttt{x}) = v$. Moreover

$$
\begin{aligned}
(\mathcal{C},\ ;MI) &\to (\mathcal{C}, MI)\\
(\mathcal{C}, [\tau]\ \mathtt{x}{:=}\mathtt{null};MI) &\to (\mathcal{C}[\mathcal{P} : \mathtt{x} \mapsto \&\mathtt{null}], MI)\\
(\mathcal{C}, [\tau]\ \mathtt{x}{:=}w;MI) &\to (\mathcal{C}[\sigma : \mathtt{x} \mapsto w], MI) \quad w \in \{\mathtt{true}, \mathtt{false}\}\\
(\mathcal{C}, [\tau]\ \mathtt{x}{:=}\mathtt{y};MI) &\to (\mathcal{C}[\mu : \mathtt{x} \mapsto \mathcal{C}(\mathtt{y})], MI) \quad \mu \in \{\sigma, \mathcal{P}, \top\mathcal{S}\}\\
(\mathcal{C}, [\tau]\ \mathtt{x}{:=}\mathtt{this};MI) &\to (\mathcal{C}[\mathcal{P} : \mathtt{x} \mapsto \top(\mathcal{S})(\mathtt{this})], MI)\\
(\mathcal{C}, [\tau]\ \mathtt{x}{:=}op(\mathtt{y}_1, \ldots, \mathtt{y}_n);MI) &\to (\mathcal{C}[\sigma : \mathtt{x} \mapsto [\![op]\!](\mathcal{C}(\mathtt{y}_1), \ldots, \mathcal{C}(\mathtt{y}_n))], MI)\\
(\mathcal{C}, [\tau]\ \mathtt{x}{:=}\mathtt{new}\ \mathrm{C}(\mathtt{y}_1, \ldots, \mathtt{y}_n);MI) &\to (\mathcal{C}[V : v \mapsto \mathrm{C}][A : (v, \mathcal{C}(\mathtt{y}_i)) \mapsto \mathtt{z}_i][\mathcal{P} : \mathtt{x} \mapsto v], MI)\\
&\quad \text{where } v \text{ is a fresh node and } \mathrm{C}.\mathcal{A} = \{\mathtt{z}_1, \ldots, \mathtt{z}_n\}\\
(\mathcal{C}, [[\tau]\ \mathtt{x}{:=}]\mathtt{y}_{n+1}.m(\mathtt{y}_1, \ldots, \mathtt{y}_n);MI) &\to (\mathcal{C}, \mathtt{push}(\{\mathtt{this} \mapsto \mathcal{C}(\mathtt{y}_{n+1}), \mathtt{z}_i \mapsto \mathcal{C}(\mathtt{y}_i)\});\\
&\qquad MI'\ [\mathtt{x}{:=}\mathtt{z};]\ \mathtt{pop};\ MI)\\
&\quad \text{with } \tau\ m(\tau_1\ \mathtt{z}_1, \ldots \tau_n\ \mathtt{z}_n)\{MI'\ [\mathtt{return}\ \mathtt{z};]\}\\
(\mathcal{C}, \mathtt{push}(\mathcal{P});MI) &\to (\mathcal{C}[\mathcal{S} : \mathtt{push}(\mathcal{P})], MI)\\
(\mathcal{C}, \mathtt{pop};MI) &\to (\mathcal{C}[\mathcal{S} : \mathtt{pop}], MI)\\
(\mathcal{C}, \mathtt{while}(\mathtt{x})\{MI'\}\ MI) &\to (\mathcal{C},\ MI'\ \mathtt{while}(\mathtt{x})\{MI'\}\ MI) \quad \text{if } \mathcal{C}(\mathtt{x}) = \mathtt{true}\\
(\mathcal{C}, \mathtt{while}(\mathtt{x})\{MI'\}\ MI) &\to (\mathcal{C},\ MI) \quad \text{if } \mathcal{C}(\mathtt{x}) = \mathtt{false}\\
(\mathcal{C}, \mathtt{if}(\mathtt{x})\{MI_{\mathtt{true}}\}\mathtt{else}\{MI_{\mathtt{false}}\}\ MI) &\to (\mathcal{C}, MI_w\ MI) \quad \text{if } \mathcal{C}(\mathtt{x}) = w \in \{\mathtt{true}, \mathtt{false}\}
\end{aligned}
$$

Fig. 3: Semantics of core Java

let $\mathcal{C}[\mathcal{S} : \mathtt{push}(\mathcal{P})]$ and $\mathcal{C}[\mathcal{S} : \mathtt{pop}]$ be notations for the memory configuration where the pointer mapping $\mathcal{P}$ has been pushed to the top of the stack and where the top pointer mapping has been removed from the top of the stack, respectively. Finally, let $\mathcal{C}[V : v \mapsto \mathrm{C}]$ denote a memory configuration $\mathcal{C}'$ whose graph contains the new node v labeled by C (i.e. $l(v) = \mathrm{C}$) and let $\mathcal{C}[A : (v, w) \mapsto \mathtt{x}]$ denote a memory configuration $\mathcal{C}'$ whose graph contains the new arrow (v, w) labeled by $\mathtt{x}$ (i.e. $i((v, w)) = \mathtt{x}$). We define $dom(\mathcal{C}) = dom(\mathcal{P}) \uplus dom(\top\mathcal{S}) \uplus dom(\sigma)$ and $[\![op]\!]$ to be the function computed by the language implementation of operator *op*.

The rules of $\to$ are defined formally in Figure 3.

4 Type system

4.1 Tiered types

The set of base types $\mathtt{T}$ is defined to be the set including a reference type C for each class name C and the special type `void` and the primitive type `boolean`. In other words, $\mathtt{T} = \{\mathtt{void}, \mathtt{boolean}\} \cup \mathbb{C}$.

Tiers are elements of the lattice $(\{\mathbf{0}, \mathbf{1}\}, \vee, \wedge)$ where $\wedge$ and $\vee$ are the greatest lower bound and least upper bound operators, respectively. The induced order, denoted $\preceq$, is such that $\mathbf{0} \preceq \mathbf{1}$. In what follows, let $\alpha, \beta, \ldots$ denote tiers in $\{\mathbf{0}, \mathbf{1}\}$. The minimum $\bigwedge_{i \in S} \alpha_i$ of a finite set is defined in a standard way.

A *tiered type* is a pair $\tau(\alpha)$ consisting of a type $\tau \in \mathtt{T}$ together with a tier $\alpha \in \{\mathbf{0}, \mathbf{1}\}$. Given a tiered type, we define the two projections π_1 and π_2 as follows: $\pi_1(\tau(\alpha)) = \tau$ and $\pi_2(\tau(\alpha)) = \alpha$.

4.2 Environments

An *attribute typing environment* $\delta_{m^{\mathrm{C}}}$ for a method m of class C maps each attribute $v \in \mathrm{C}.\mathcal{A}$ to a tiered type. A *local variable and parameter typing environment* δ_g maps each non-attribute variable in $\mathbb{V}$ to a tiered type. A *typing environment* Δ is a list containing the unions $\delta_{m^{\mathrm{C}}} \biguplus \delta_g$ for each attribute typing environment and the local variable and parameter typing environment. Let $\Delta(m^{\mathrm{C}})$ denote $\delta_{m^{\mathrm{C}}} \biguplus \delta_g$. A *contextual typing environment* $\Gamma = (m^{\mathrm{C}}, \Delta)$ is a pair consisting in a method and a typing environment. The method m^{C} indicates in which context the attributes should be typed, in other words the $\delta_{m^{\mathrm{C}}}$ is considered. We will write (ϵ, Δ) when the context is empty. $\Delta(\epsilon)$ represents the local variable and parameter typing environment δ_g.

Intuitively, tier **0** will be used to type variables whose corresponding stored values might increase during a computation whereas tier **1** will be used to type variables used in the guard of a while loop or as a recursive argument of a method call. Consequently, the values stored in a tier **1** variable will not be allowed to increase during a computation. Given a typing environment $\Delta = (..., \delta, ...)$ and a tier α, let Δ_α be the list of $(..., \delta^\alpha, ...)$ where δ^α is the restriction of δ to variables or attributes of tier α, that is such that $\pi_2(\delta(\mathtt{x})) = \alpha$).

4.3 Well-typed programs

Operator signature The language is restricted to operators whose return type is `boolean`. An operator of arity n comes equipped with a signature of the shape $\tau_1 \times \cdots \times \tau_n \to \mathtt{boolean}$, fixed by the language implementation. In the type system, the notation $op :: \tau_1 \times \cdots \times \tau_n \to \mathtt{boolean}$ denotes that op has signature$\tau_1 \times \cdots \times \tau_n \to \mathtt{boolean}$.

Judgments Expressions and instructions will be typed using tiered types whereas constructors and methods of arity n have types of the shape: $\tau_1(\alpha_1) \times \ldots \times \tau_n(\alpha_n) \to \tau(\alpha)$ and $\mathrm{C}(\beta) \times \tau_1(\alpha_1) \times \ldots \times \tau_n(\alpha_n) \to \tau(\alpha)$ respectively. Given a contextual typing environment Γ, there are four kinds of typing judgments:

- The judgment $\Gamma \vdash E : \tau(\alpha)$ means that expression E corresponds to values of tiered type $\tau(\alpha)$.
- The judgment $\Gamma \vdash I : \mathtt{void}(\alpha)$ is similar but the type is enforced to be `void`, meaning that instructions have no return value.
- The judgment $\Gamma \vdash K_{\mathrm{C}} : \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \mathrm{C}(\mathbf{0})$ enforces the output of a constructor to be of the correct type C and to be of tier **0**, this important tiering restriction will prevent object instantiation in variables of tier **1**.
- The last judgment $\Gamma \vdash M_{\mathrm{C}} : \mathrm{C}(\beta) \times \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \tau(\alpha)$ for methods is similar but unrestricted.

Well-typedness Let us now introduce the notion of well-typed program. Intuitively, a well-typed program has an executable whose initialization instruction is only constrained by types and whose computational instruction is both constrained on types and tiers. The type system propagates these constraints on all the classes, methods and instructions used within these instructions.

Definition 6 (Well-typed program). *Given a program of executable Exe and a typing variable environment $\Gamma = (\epsilon, \Delta)$, the judgment $\Gamma \vdash Exe : \diamond$ means that the program is well-typed wrt Γ.*

4.4 Typing rules

Expressions The typing rules for expressions are provided in Figure 4.

Rules *(True)*, *(False)*, *(Null)* and *(Var)* are self-explanatory.

The *(Self)* rule explicits that the self reference `this` belongs to class C and has a tier lower than the minimum of the attributes' tiers under the typing environment $\Delta(m^{\mathrm{C}})$, denoted by $\bigwedge \Delta(m^{\mathrm{C}})$ and defined formally by:

$$\bigwedge \Delta(m^{\mathrm{C}}) = \bigwedge_{\mathtt{x} \in \mathrm{C}.\mathcal{A}} (\pi_2(\Delta(m^{\mathrm{C}})(\mathtt{x}))).$$

This entails that an object of tier **1** has no attribute of tier **0**, in other words, no arrow will go from a node corresponding to tier **1** to a node of tier **0**.

Rule *(Op)* describes how to type an operator applied to n arguments. The arguments must be of types corresponding to the operator signature. The operands must be of the same tier α which will also be the tier of the whole expression. It prevents information to flow from a tier to the other.

Rule *(New)* describes the typing of object instantiation. It checks that the constructor arguments have tiered types $\tau_i(\beta_i)$ of the same types τ_i and of tier not lower than the admissible tiers α_i in the constructor typing judgment. Note that the new instance has tier **0** since its creation makes the memory grow.

Rule *(Call)* represents how to type method calls of the shape $E.m(E_1, \ldots, E_n)$. First, we check that the tiered type $\mathrm{C}(\beta)$ of the self reference in the method m matches the tiered type of the instance E. Simultaneously, the current method in the contextual typing environment is updated to m^{C}. We check that the arguments' tiered-types agree with the parameters' tiered-types in m's signature. An important point to stress is that the tier of the evaluated expression (or instruction) in a method call matches the tier of the return variable in the method, hence avoiding forbidden information flows.

Instructions The typing rules for instructions are provided in Figure 5.

Rule *(Ass)* explains how to type an assignment: it is an instruction, hence of type `void`. It is only possible to assign an expression E to a variable `x` if both the types match and the tier β of E is higher than the tier α of `x`. The tier of the instruction will be α. This rule implies that information may flow from tier **1** to tier **0** but not the contrary.

$$\frac{}{(m^{\mathrm{C}}, \Delta) \vdash \texttt{true} : \texttt{boolean}(\mathbf{1})}\ \textit{(True)} \qquad \frac{}{(m^{\mathrm{C}}, \Delta) \vdash \texttt{false} : \texttt{boolean}(\mathbf{1})}\ \textit{(False)}$$

$$\frac{}{(m^{\mathrm{C}}, \Delta) \vdash \texttt{null} : \mathrm{C}(\mathbf{1})}\ \textit{(Null)} \qquad \frac{\alpha \preceq \bigwedge \Delta(m^{\mathrm{C}})}{(m^{\mathrm{C}}, \Delta) \vdash \texttt{this} : \mathrm{C}(\alpha)}\ \textit{(Self)}$$

$$\frac{\Delta(m^{\mathrm{C}})(\mathtt{x}) = \tau(\alpha)}{(m^{\mathrm{C}}, \Delta) \vdash \mathtt{x} : \tau(\alpha)}\ \textit{(Var)} \qquad \frac{\forall i,\ (m^{\mathrm{C}}, \Delta) \vdash E_i : \tau_i(\alpha) \qquad op :: \tau_1 \times \cdots \times \tau_n \to \texttt{boolean}}{(m^{\mathrm{C}}, \Delta) \vdash op(E_1, \ldots, E_n) : \texttt{boolean}(\alpha)}\ \textit{(Op)}$$

$$\frac{\forall i\ (m^{\mathrm{C}}, \Delta) \vdash E_i : \tau_i(\beta_i) \quad \alpha_i \preceq \beta_i \quad (\epsilon, \Delta) \vdash \mathrm{C}(\ldots \tau_i\ \mathtt{y}_i \ldots)\{\ldots \mathtt{x}_i := \mathtt{y}_i; \ldots\} : \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \mathrm{C}(\mathbf{0})}{(m^{\mathrm{C}}, \Delta) \vdash \texttt{new}\ \mathrm{C}(E_1, \ldots, E_n) : \mathrm{C}(\mathbf{0})}\ \textit{(New)}$$

$$\frac{\forall i\ (m_1^{\mathrm{C}_1}, \Delta) \vdash E_i : \tau_i(\beta_i) \quad \alpha_i \preceq \beta_i \quad (m^{\mathrm{C}}, \Delta) \vdash m : \mathrm{C}(\beta) \times \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \tau(\alpha) \quad (m_1^{\mathrm{C}_1}, \Delta) \vdash E : \mathrm{C}(\beta)}{(m_1^{\mathrm{C}_1}, \Delta) \vdash E.m(E_1, \ldots, E_n) : \tau(\alpha)}\ \textit{(Call)}$$

Fig. 4: Type system for core Java: Expressions

$$\frac{(m^C, \Delta) \vdash \mathtt{x} : \tau(\alpha) \quad (m^C, \Delta) \vdash E : \tau(\beta) \quad \alpha \preceq \beta}{(m^C, \Delta) \vdash [\tau]\ \mathtt{x}{:=}E; : \texttt{void}(\alpha)}\ \textit{(Ass)} \qquad \frac{(m^C, \Delta) \vdash I : \texttt{void}(\alpha) \quad \alpha \preceq \beta}{(m^C, \Delta) \vdash I : \texttt{void}(\beta)}\ \textit{(Sub)}$$

$$\frac{\forall i,\ (m^C, \Delta) \vdash I_i : \texttt{void}(\alpha_i)}{(m^C, \Delta) \vdash I_1\ I_2 : \texttt{void}(\alpha_1 \vee \alpha_2)}\ \textit{(Seq)} \qquad \frac{(m^C, \Delta) \vdash E : \texttt{boolean}(\mathbf{1}) \quad (m^C, \Delta) \vdash I : \texttt{void}(\mathbf{1})}{(m^C, \Delta) \vdash \texttt{while}(E)\{I\} : \texttt{void}(\mathbf{1})}\ \textit{(Wh)}$$

$$\frac{}{(m^C, \Delta) \vdash\ ; : \texttt{void}(\mathbf{0})}\ \textit{(Skip)} \qquad \frac{(m^C, \Delta) \vdash E : \texttt{boolean}(\alpha) \quad \forall i,\ (m^C, \Delta) \vdash I_i : \texttt{void}(\alpha)}{(m^C, \Delta) \vdash \texttt{if}(E)\{I_1\}\texttt{else}\{I_2\} : \texttt{void}(\alpha)}\ \textit{(If)}$$

Fig. 5: Type system for core Java: Instructions

$$\frac{\forall i,\ (\epsilon, \Delta) \vdash \mathtt{y}_i : \tau_i(\alpha_i)}{(\epsilon, \Delta) \vdash \mathrm{C}(\tau_1\ \mathtt{y}_1, \ldots, \tau_n\ \mathtt{y}_n)\{\mathtt{x}_1 := \mathtt{y}_1; \ldots \mathtt{x}_n := \mathtt{y}_n\} : \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \mathrm{C}(\mathbf{0})}\ \textit{(K}_C\textit{)}$$

$$\frac{(m^{\mathrm{C}}, \Delta) \vdash \texttt{this} : \mathrm{C}(\beta) \qquad \forall i,\ (m^{\mathrm{C}}, \Delta) \vdash \mathtt{x}_i : \tau_i(\alpha_i) \qquad (m^{\mathrm{C}}, \Delta) \vdash I : \texttt{void}(\alpha)}{(\epsilon, \Delta) \vdash \texttt{void}\ m(\tau_1\ \mathtt{x}_1, \ldots, \tau_n\ \mathtt{x}_n)\{I\} : \mathrm{C}(\beta) \times \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \texttt{void}(\alpha)}\ \textit{(M}_C^{\texttt{void}}\textit{)}$$

$$\frac{(m^{\mathrm{C}}, \Delta) \vdash \texttt{this} : \mathrm{C}(\beta) \quad \forall i,\ (m^{\mathrm{C}}, \Delta) \vdash \mathtt{x}_i : \tau_i(\alpha_i) \quad (m^{\mathrm{C}}, \Delta) \vdash \mathtt{x} : \tau(\alpha) \quad (m^{\mathrm{C}}, \Delta) \vdash I : \texttt{void}(\alpha)}{(\epsilon, \Delta) \vdash \tau\ m(\tau_1\ \mathtt{x}_1, \ldots, \tau_n\ \mathtt{x}_n)\{I\ \texttt{return}\ \mathtt{x};\} : \mathrm{C}(\beta) \times \tau_1(\alpha_1) \times \cdots \times \tau_n(\alpha_n) \to \tau(\alpha)}\ \textit{(M}_C\textit{)}$$

$$\frac{(\epsilon, \Delta) \vdash I : \texttt{void}(\mathbf{1}) \quad \forall i,\ (\epsilon, \Delta) \vdash \mathtt{x}_i : \tau_i(\alpha_i) \quad \forall i,\ (\epsilon, \Delta) \vdash E_i : \tau_i(\beta_i)}{(\epsilon, \Delta) \vdash \mathrm{Exe}\{\texttt{main}()\{\tau_1\ \mathtt{x}_1 := E_1; \ldots; \tau_n\ \mathtt{x}_n := E_n; I\}\} : \diamond}\ \textit{(Main)}$$

Fig. 6: Type system for core Java: Constructors, methods and executable

Rule *(Sub)* is a sub-typing rule allowing to elevate an instruction of tier α to tier β with $\alpha \preceq \beta$.

Rule *(Seq)* types the sequence of two instructions I_1 and I_2. The sequence's tier will be the maximum of the tiers of I_1 and I_2.

Rule *(If)* describes the typing discipline for a $\texttt{if}(E)\{I_1\}\texttt{else}\{I_2\}$ statement. E needs to be a boolean expression of tier α. I_1 and I_2 are instructions, hence of type `void`, with the same tier α. This prevents assignments of tier **1** variables in the instructions I_1 and I_2 to be controlled by a tier **0** expression.

Rule *(Skip)* is standard.

Rule *(Wh)* is the most important typing rule as it will constrain the use of while loops. In a statement $\texttt{while}(E)\{I\}$, the guard of the loop E must be a boolean expression of tier **1** so that the guard is controlled. The instruction I, of type `void`, has to be of tier **1** since we expect the guard variables to be modified (i.e. assigned to). The whole statement is an instruction of type `void` and tier **1**.

Methods, constructors and executable The typing rules for constructors and methods are provided in Figure 6.

Rule *(K_C)* describes the typing of a constructor definition. Constructors are of fixed form, so the only thing to check is that the parameters are of the desired tiered types. As explained in Rule *(New)* the output tier can only be **0**.

Rules *($M_C^{\texttt{void}}$)* and *(M_C)* show how to type method definitions. In both cases, the types and number of parameters need to match the method signature, the instruction I in the body of the method needs to be of type $\texttt{void}(\alpha)$, i.e. the tier matches the output tier so that there is no forbidden information flow.

Finally, typing an executable is done through the rule *(Main)* and consists in verifying that the initialization instruction respects types and that the computational instruction (denoted by instruction I) is of tier **1**. Notice that no tier constraints are checked in the initialization instruction: this means that we do not control the complexity of this latter instruction ; the main reason for this choice is that this instruction is considered to be building the program input. In opposition, the computational instruction I is considered to be the computational part of the program and has to respect the tiering discipline.

5 Upper bound on the stack size and the heap size

5.1 Definitions

In this section, we state our main result showing that well-formed and typed programs have both pointer stack size and pointer graph size bounded polynomially by the input size under termination and safety assumptions. Moreover, precise upper bounds can be extracted. For that purpose, we need to define the notion of size for pointer stack, pointer graph and memory configuration.

Definition 7 (Sizes).

- *The size of a pointer graph $\mathcal{G}_\mathcal{P}$ is defined to be the number of nodes in $\mathcal{G}$ and denoted $|\mathcal{G}_\mathcal{P}|$.*

– *The size of a pointer stack $\mathcal{S}_\mathcal{G}$ is defined to be the number of pointer mappings in the stack $\mathcal{S}$ and denoted $|\mathcal{S}_\mathcal{G}|$.*
– *The size of a memory configuration $\langle \mathcal{G}, \mathcal{P}, \mathcal{S}, \sigma \rangle$ is equal to $|\mathcal{G}_\mathcal{P}| + |\mathcal{S}_\mathcal{G}| + |dom(\mathcal{P})| + |dom(\sigma)|$.*

Since a pointer graph contains both references to the objects (the nodes) and references to the attribute instances (the arrows), it would make sense to bound both the number of nodes and the number of arrows in order to control the heap-space, for a practical application. Notice that the outdegree of a node is bounded by a constant of the program (the maximum number of attributes in a class) and, consequently, bounding the number of nodes is sufficient to obtain a big O bound. The size of a pointer stack is very close to the size of the Java Virtual Machine stack since it counts the number of nested method calls.

Given two methods M_C and $M'_{\mathrm{C}'}$ of respective signatures s and s' and respective names m and m', define the relation $\sqsubset$ on method signatures by $s \sqsubset s'$ if m' is called in M_C, i.e. in the body of M_C (this check is fully static as long as we do not consider inheritance). Let $\sqsubset^+$ be its transitive closure. A method of signature s is *recursive* if $s \sqsubset^+ s$ holds. Given two method signatures s and s', $s \equiv s'$ holds if both $s \sqsubset^+ s'$ and $s' \sqsubset^+ s$ hold. Given a signature s, the equivalence class $[s]$ is defined as usual by $[s] = \{s' \mid s' \equiv s\}$. When the signature s of a given method M_C of name m is clear from the context, we will write $[m]$ as an abuse of notation for $[s]$ and say that M_C is a recursive method. Finally, we write $s \sqsubsetneq^+ s'$ if $s \sqsubset^+ s'$ holds and $s' \sqsubset^+ s$ does not hold.

The notion of level of a meta-instruction is introduced to compute an upper bound on the number of recursive steps for a method call evaluation.

Definition 8 (Level). *Let the level λ of a method signature be defined as follows:*

– $\lambda(s) = 1$ *if* $s \notin [s]$
– $\lambda(s) = \max\{1 + \lambda(s') \mid s \sqsubsetneq^+ s'\}$ *otherwise.*

By abuse of notation, we will write $\lambda(m)$ when the signature of m is clear from the context. For a given program, we denote the maximal level of a method by λ.

The notion of intricacy corresponds to the number of nested `while` loops in a meta-instruction and will be used to compute the requested upper bounds.

Definition 9 (Intricacy). *Let the intricacy ν of a meta-instruction be defined as follows:*

– $\nu(;) = \nu(\mathtt{pop};) = \nu(\mathtt{push}(\mathcal{P});) = \nu(\mathtt{x}{:=}ME;) = 0$
– $\nu(MI\ \ MI') = \max(\nu(MI), \nu(MI'))$
– $\nu(\mathtt{if(x)\{}MI\mathtt{\}else\{}MI'\mathtt{\}}) = \max(\nu(MI), \nu(MI'))$
– $\nu(\mathtt{while(x)\{}MI\mathtt{\}}) = 1 + \nu(MI)$

Moreover, let ν be the maximal intricacy of a meta-instruction within a given program.

Notice that both intricacy ν and level λ are bounded by the size of their corresponding program.

5.2 Safety restriction on recursive methods

Now we put some aside restrictions on recursive methods to ensure that their computations remain polynomially bounded. Recursive methods will be restricted to have only one recursive call and no while loop in their body (to prevent exponential growth) and must have tier **1** input (as the guard of a `while`) and output (to prevent a recursive dependence on a tier **0** variable).

Definition 10 (Safety). *A well-typed program with respect to a typing environment Δ is* safe *if for each recursive method $M_C = \tau\ m(\ldots)\{MI\ [\texttt{return}\ \boldsymbol{x};]\}$:*

- *there is exactly one call to some $m' \in [m]$ in MI,*
- *there is no while loop inside MI, i.e. $\nu(MI) = 0$,*
- *and the following judgment can be derived:*

$$(\epsilon, \Delta) \vdash M_C : C(\mathbf{1}) \times \tau_1(\mathbf{1}) \times \cdots \times \tau_n(\mathbf{1}) \to \tau(\mathbf{1}).$$

Remark 1. A program is safe wrt a typing environment Δ iff its flattened version is safe wrt a typing environment $\Delta' \supseteq \Delta$.

5.3 Main results

The presented type system allows us to infer polynomial upper bounds on the stack and the heap of safe programs. In the following theorem, we use $n_{\mathbf{1}}$ to denote the number of variables and attributes of tier **1** in the whole program, that is $\sum \#dom(\delta_i^{\mathbf{1}})$ for a typing environment $\Delta = (..., \delta_i, ...)$

Theorem 1. *If a core Java program of computational instruction I is safe wrt to typing environment Δ and terminates on input $\mathcal{C}$ then for each memory configuration $\mathcal{C}'$ and meta-instruction MI s.t. $(\mathcal{C}, \overline{I}) \to^* (\mathcal{C}', MI)$ we have:*

$$|\mathcal{C}'| = O(|\mathcal{C}|^{n_{\mathbf{1}}((\nu+1)\lambda)}).$$

In other words, if $\mathcal{C}' = \langle \mathcal{G}, \mathcal{P}, \mathcal{S}, \sigma \rangle$ then both $|\mathcal{G}_{\mathcal{P}}|$ and $|\mathcal{S}_{\mathcal{G}}|$ are in $O(|\mathcal{C}|^{n_{\mathbf{1}}((\nu+1)\lambda)})$.

As a corollary, if the program terminates on all input configurations, then we may infer a polynomial time upper bound on its execution time.

Corollary 1. *If a core Java program of computational instruction I is safe wrt to typing environment Δ and terminates on input $\mathcal{C}$ then it does terminate in time $O(|\mathcal{C}|^{n_{\mathbf{1}}((\nu+1)\lambda)})$.*

Another corollary is that tier **1** variables remain polynomially bounded without assuming termination. It means we can guarantee security properties on the data stored in such variables even if we are unable to prove program termination.

Corollary 2. *If a core Java program of computational instruction I is safe wrt to typing environment Δ then, on input $\mathcal{C}$, for each memory configuration $\mathcal{C}'$, meta-instruction MI s.t. $(\mathcal{C}, \overline{I}) \to^* (\mathcal{C}', MI)$ and variable $\boldsymbol{x} \in dom(\delta_g^{\mathbf{1}})$ we have:*

$$|\mathcal{C}'(\boldsymbol{x})| = O(|\mathcal{C}|^{n_{\mathbf{1}}((\nu+1)\lambda)}).$$

where $|\mathcal{C}'(\boldsymbol{x})|$ denotes the size of the subgraph of nodes reachable from node $\mathcal{C}'(\boldsymbol{x})$ whenever $\boldsymbol{x}$ is of reference type.

Another direct result is that our characterization is complete with respect to the class of functions computable in polynomial time as a direct consequence of Marion's result [22] since both our language and type system can be viewed as an extension of the considered imperative language. This means that our type system has a good expressivity.

Finally, we show the decidability of type inference wrt our type system:

Proposition 1 (Type inference). *Deciding if there exists a variable typing environment Γ such that typing rules are satisfied can be done in time linear in the size of the program.*

5.4 An illustrating example

Let us apply our framework to a simple class `BList` for encoding binary integers as binary lists (with the least significant bit in head). Tiers are made explicit in the code: The notation $\mathtt{x}^{\alpha}$ means that `x` has tier α under the considered contextual typing environment Γ, i.e. $\Gamma \vdash \mathtt{x} : \tau(\alpha)$, for some τ, whereas $I : \alpha$ means that $\Gamma \vdash I : \mathtt{void}(\alpha)$.

```
1  BList {
2    boolean value;
3    BList queue;
4
5    BList(boolean v, BList q) {
6      value = v;
7      queue = q;
8    }
```

The constructor BList can be typed by boolean(α) $\times$ Blist(β) $\rightarrow$ BList($\mathbf{0}$), with $\alpha,\ \beta \in \{\mathbf{0}, \mathbf{1}\}$, depending on the typing environment, by rule (K_{C}).

```
9    BList getQueue() { return queue; }
```

The method getQueue can be typed by BList($\mathbf{1}$) $\rightarrow$ BList($\mathbf{1}$) or BList($\mathbf{0}$) $\rightarrow$ BList($\mathbf{1}$) if $\Delta(\mathrm{getQueue}^{\mathrm{BList}})(\mathrm{queue})=\mathbf{1}$ and by BList($\mathbf{0}$) $\rightarrow$ BList($\mathbf{0}$) if $\Delta(\mathrm{getQueue}^{\mathrm{BList}})(\mathrm{queue})=\mathbf{0}$, by rules ($M_{\mathrm{C}}$) and (Self).

The type BList($\mathbf{1}$) $\rightarrow$ BList($\mathbf{0}$) is prohibited by rule (Self) since the tier of the current object $\mathbf{1}$ has to be lower than the minimum tier of its arguments $\mathbf{0}$.

```
10   void setQueue(BList q) { queue = q; }
```

The method setQueue can be given the types BList($\mathbf{1}$) $\times$ BList($\mathbf{1}$) $\rightarrow$ `void`($\mathbf{1}$), BList($\mathbf{0}$) $\times$ BList($\mathbf{1}$) $\rightarrow$ `void`(α), or BList($\mathbf{0}$) $\times$ BList($\mathbf{0}$) $\rightarrow$ `void`(α), $\alpha \in \{\mathbf{0}, \mathbf{1}\}$.

The input type BList($\mathbf{1}$) $\times$ BList($\mathbf{0}$) is prohibited since the tier of parameter q ($\mathbf{0}$) has to be greater than the tier of the attribute queue, by rule (Ass); but this latter tier has to be greater than the tier of the current object ($\mathbf{1}$), by rule (Self). Finally, the output tier corresponds to the tier of the typed instruction by rule (M_{C}). Consequently, it also corresponds to the tier of the attribute queue by rule (Ass) and is enforced to be $\mathbf{1}$ when the current object has type $\mathbf{1}$, by the rule (Self). Notice that it can be $\mathbf{1}$ whenever the object current tier is $\mathbf{0}$ using the subtyping rule (Sub).

```
11    boolean getValue() { return value; }
```

getValue can be given the types BList($\mathbf{1}$) $\to$ boolean($\mathbf{1}$), BList($\mathbf{0}$) $\to$ boolean($\mathbf{1}$) or BList($\mathbf{0}$) $\to$ boolean($\mathbf{0}$) (the explanations are the same than for the getQueue getter).

```
12    BList double() {
13      BList n^0 = new BList(false, this);
14      return n^0;
15    }
```

The method double can be typed by BList($\mathbf{0}$) $\to$ BList($\mathbf{0}$) or BList($\mathbf{1}$) $\to$ BList($\mathbf{0}$). Indeed the local variable n is enforced to be of tier $\mathbf{0}$ by a combination of rules (K_{C}) and (Ass). Consequently, the method output type is BList($\mathbf{0}$) since it has to match the type of the returned variable. Finally, there is no constraint on the current object admissibile types since it is left unchanged by the method.

```
16    void decrement() {
17      if (value^1 == true or value^1 == null) {
18        value^1 = false; :1
19      } else {
20        if (queue^1 != null) {
21          value = true;
22          queue^1.decrement(); :1
23        } else { value^1 = false; :1 }
24      }
25    } :1
```

The method decrement is recursive. Consequently, the type BList($\mathbf{1}$) $\to$ `void`($\mathbf{1}$) is mandatory by safety. This enforces the tier of the each attribute to be $\mathbf{1}$ by rule (Self). Finally, the method body can be typed using a combination of rules (Ass), (If) and (Call).

```
26    void concat(BList other^1) {
27      BList o^1 = this^1;:1
28      while (o^1.getQueue() != null) { o^1 = o^1.getQueue();:1 }
29      o^1.setQueue(other^1);:1
30    }
```

In the concat method, the presence of the method call o.getQueue() in the guard of the while loop enforces its ouput tier to be $\mathbf{1}$ by rules (Wh), (Null) and (Op). Consequently, the type of getQueue has to be BList($\mathbf{1}$) $\to$ BList($\mathbf{1}$) under the considered contextual typing environment (see the admissible types of getQueue). It also enforces the object o to be of tier $\mathbf{1}$. Consequently, the current object `this` is also enforced to be of tier $\mathbf{1}$ by rule (Ass) and the tier of the parameter other is enforced to be $\mathbf{1}$ (see the setQueue admissible types). Consequently, the only admissible type for concat is BList($\mathbf{1}$) $\times$ BList($\mathbf{1}$) $\to$ `void`($\mathbf{1}$), using rule (Seq).

```
31    boolean isEqual(BList other^1) {
32      boolean res^0 = true; :1 //using (Sub)
33      BList b1^1 = this^1; :1
```

```
34      BList b2^1 = other^1; :1
35      while (b1^1 != null && b2^1 != null) {
36        if (b1^1.getValue() != b2^1.getValue()) { res^0 = false;:1 }
37        b1^1 = b1^1.getQueue();:1
38        b2^1 = b2^1.getQueue();:1
39      }
40      if (b1^1 != null || b2^1 != null) { res^0 = false;:1 }
41      return res^0;
42    }
43  }
```

The local variables b1 and b2 are enforced to be of tier **1** by rule (Wh). Consequently, this and other are also of tier **1** using twice rule (Ass). Consequently, the methods getValue and getQueue will be typed by BList(**1**) $\to$ boolean(**1**) and BList(**1**) $\to$ BList(**1**), respectively. Finally, the local variable can be given the type boolean(**1**) or boolean(**0**) (in this latter case, the subtyping rule (Sub) will ne needed) and, consequently, the admissible types for isEqual are BList(**1**) $\times$ BList(**1**) $\to$ boolean(α), $\alpha \in \{\mathbf{0}, \mathbf{1}\}$.

6 Conclusion

This work presents a simple type-system (it can be checked in linear time) that provides explicit polynomial upper bounds on the heap and stack size of an object oriented program allowing (recursive) method calls. As the system is purely static, the bounds are not as tight as may be desirable. It would indeed be possible to refine the framework to obtain a better exponent at the price of a non-uniform formula (for example not considering all tier **1** variables but only those modified in each while loop or recursive method would reduce the computed complexity). OO features, such as inheritance, abstract classes, interfaces and static attributes and methods, were not considered here, but we claim that they can also be treated by our analysis. Moreover, constructs that breaks the control flow like break, return and continue can also be considered in our fragment (they have to be constrained to be of tier **1** so that if such an instruction is to be executed, then we know that it does not depend on tier **0** expressions). Also note that the safety condition can be alleviated on recursive methods by ensuring that only one recursive call is reachable in the execution of the method body.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Costa: Design and implementation of a cost and termination analyzer for java bytecode. In: FMCO. LNCS, vol. 5382, pp. 113–132 (2007)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theor. Comput. Sci. 413(1), 142–159 (2012)
3. Bellantoni, S., Cook, S.: A new recursion-theoretic characterization of the poly-time functions. Comput. Complex. 2, 97–110 (1992)

4. Ben-Amram, A.M.: Size-change termination, monotonicity constraints and ranking functions. Log. Meth. Comput. Sci. 6(3) (2010)
5. Ben-Amram, A.M., Genaim, S., Masud, A.N.: On the termination of integer loops. In: VMCAI. LNCS, vol. 7148, pp. 72–87 (2012)
6. Beringer, L., Hofmann, M., Momigliano, A., Shkaravska, O.: Automatic certification of heap consumption. In: LPAR. LNCS, vol. 3452, pp. 347–362 (2004)
7. Cachera, D., Jensen, T., Pichardie, D., Schneider, G.: Certified memory usage analysis. In: FM 2005: Formal Methods. LNCS, vol. 3582, pp. 91–106 (2005)
8. Chin, W., Nguyen, H., Qin, S., Rinard, M.: Memory usage verification for OO programs. In: Static Analysis, SAS 2005. pp. 70–86 (2005)
9. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond safety. In: CAV. LNCS, vol. 4144, pp. 415–426 (2006)
10. Gulwani, S.: Speed: Symbolic complexity bound analysis. In: CAV. LNCS, vol. 5643, pp. 51–62 (2009)
11. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: Speed: precise and efficient static estimation of program computational complexity. In: POPL. pp. 127–139. ACM (2009)
12. Hainry, E., Marion, J.Y., Péchoux, R.: Type-based complexity analysis for fork processes. In: FOSSACS. LNCS, vol. 7794, pp. 305–320 (2013)
13. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: POPL. pp. 185–197. ACM (2003)
14. Hofmann, M., Jost, S.: Type-based amortised heap-space analysis. In: ESOP. LNCS, vol. 3924, pp. 22–37 (2006)
15. Hofmann, M., Rodriguez, D.: Efficient type-checking for amortised heap-space analysis. In: CSL. LNCS, vol. 5771, pp. 317–331 (2009)
16. Hofmann, M., Schöpp, U.: Pointer programs and undirected reachability. In: LICS. pp. 133–142. IEEE Computer Society (2009)
17. Hofmann, M., Schöpp, U.: Pure pointer programs with iteration. ACM Trans. Comput. Log. 11(4) (2010)
18. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight java: a minimal core calculus for java and GJ. ACM Trans. Program. Lang. Syst. 23(3), 396–450 (2001)
19. Jones, N.D., Kristiansen, L.: A flow calculus of *wp*-bounds for complexity analysis. ACM Trans. Comput. Log. 10(4) (2009)
20. Jost, S., Hammond, K., Loidl, H.W., Hofmann, M.: Static determination of quantitative resource usage for higher-order programs. In: POPL. pp. 223–236 (2010)
21. Leivant, D., Marion, J.Y.: Lambda calculus characterizations of poly-time. Fundam. Inform. 19(1/2), 167–184 (1993)
22. Marion, J.Y.: A type system for complexity flow analysis. In: LICS. pp. 123–132 (2011)
23. Moyen, J.Y.: Resource control graphs. ACM Trans. Comput. Logic 10(4) (2009)
24. Niggl, K.H., Wunderlich, H.: Certifying polynomial time and linear/polynomial space for imperative programs. SIAM J. Comput. 35(5), 1122–1147 (2006)
25. Podelski, A., Rybalchenko, A.: Transition predicate abstraction and fair termination. In: POPL. pp. 132–144. ACM (2005)
26. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Computer Security 4(2/3), 167–188 (1996)